Electronic Notes in Theoretical Computer Science 64 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume64.html 18 pages

Timed Term Rewrite Systems

Jérémie Blanc and Rachid Echahed

Laboratoire LEIBNIZ – Institut IMAG, CNRS
46, avenue Félix Viallet, F-38031 Grenoble, France
Tel: (+33) 4 76 57 46 73; Fax: (+33) 4 76 57 46 02
Jeremie.Blanc@imag.fr Rachid.Echahed@imag.fr

Abstract

We propose a new class of rewrite systems that are conservative extensions of first-order conditional term rewrite systems together with time features such as clocks, signals, timed terms, timed atoms and timed rules. We define first timed term rewrite systems by their syntax and illustrate them through some examples. We provide then a semantics based on labelled transition systems. Finally, we show how our framework compares to related work.

1 Introduction

Term rewrite systems have been widely investigated during the last years [5,11,6]. They are the bases of many modern declarative languages, theorem provers and program validation techniques. The main motivation of this work comes from the observation that term rewrite systems fail to specify in a natural way real-world applications where *time* is involved. Indeed, consider for instance the boolean operator Alarm with the following profile device_sort $\longrightarrow$ bool such that Alarm(device) is true whenever device has been broken for the last ten seconds. Unfortunately, such simple program cannot be described rigorously using classical rewrite systems.

In this paper, we introduce timed term rewrite systems as a new class of rewrite systems which allows one to specify declaratively applications where the notion of time, should it be qualitative or quantitative, is involved. Our approach is new and departs from the proposals already made in order to add time into some declarative languages such as `tcc` [16], `Templog` [1], and `Chronolog` [19]. Roughly speaking, a timed rewrite system is provided with user-defined and incoming signals like in synchronous languages, e.g. [9]. A signal is a stream of pairs (ticks, values) that happen over time. We assume given a canonical signal or clock, noted Ref which serves as a reference for other signals. In general, operator definitions within a timed rewrite system or program $\mathcal{P}$ may depend on time. Thus, at each instant t, of Ref, a

new rewrite system consisting of operator definitions is updated in the same way as in data-flow languages. We obtain then, a stream of rewrite systems, $\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_t, \ldots$. At any instant t, $\mathcal{F}_t$ constitutes a classical rewrite system which may be used as usual, e.g., simplification or resolution of formulas.

In order to make easier the description of such timed rewrite systems, we enrich first-order terms by a temporal operator *at* which is used to refer to past definitions. For example the expression $at(-3\ seconds)f(0) + f(1)$ is meant to add the value of $f(0)$ three seconds before now and the value of $f(1)$ at the current instant. On the other hand, we also enrich classical atoms by allowing the use of the box $\Box$ and diamond $\Diamond$ modal operators over intervals. A timed atom $\Box_I B$, respectively $\Diamond_I B$, holds iff the atom B holds at every instant, respectively at one instant at least, of the time interval I. The rewrite rules we consider in this paper have the following general shape $lhs \rightarrow trhs \Leftarrow Body$ **when** $Tail$ where lhs (resp. $trhs$) is a term (resp. timed term) representing the left-hand side (resp. right-hand side) of the rule, $Body$ and $Tail$ are conjunctions of possibly timed atoms or equations. The role of tails is to filter at every instant, t, the rewrite rules that constitute the rewrite system $\mathcal{F}_t$.

The rest of the paper is organized as follows. The next section introduces the class of timed rewrite systems and provide sample examples illustrating our framework. Then, we propose a semantics of our programs based on labelled transition systems. In section 4, we compare our proposal to related work. Section 5 points some issues related to operational semantics and concludes the paper.

2 Timed Rewrite Systems

We define a *timed rewrite system* or program as a pair $\mathcal{P} = < \Sigma , R >$ where Σ is a timed first order signature and R is a set of timed conditional rewrite rules.

A *timed first order signature* is a triple $< S , \mathcal{S} , \Omega >$ where S is a set of sorts, $\mathcal{S}$ is an S^+-sorted family of signals and Ω is an S^+-sorted family of timed operators. S^+ stands for the set of non empty strings over S. We assume that S contains at least the sort of booleans, bool, and the sort of naturals, nat.

The meaning of a signal in our approach is very close to the ones introduced in some synchronous languages, e.g., [9,7]. A *signal* s of sort $s_1 , \ldots , s_n , s$, noted $\mathrm{s} : s_1 , \ldots , s_n \longrightarrow s$ is a stream of operator denotations. These denotations are associated implicitly to some instants over time. These instants constitute a set called ticks of s. Hence, our notion of time is linear and multiform. Every signal s is associated to a boolean operator noted !s which is equal to true on the ticks of the considered signal s.

The family $\mathcal{S}$ contains a distinguished signal we call *reference signal* of profile Ref : $\longrightarrow$ nat. More precisely, Ref is the finest signal i.e. for all signals s in $\mathcal{S}$, the ticks of s are mapped into those of Ref. At the beginning,

the value of Ref is 0, then it increments by one at each instant. Its value may be useful as a clock which gives the number of the current instant.

Starting from a timed signature, we introduce the notion of *timed terms* in order to take into account the "temporization" of operators thanks to the presence of signals. The definition of an operator, say ω, may change at every instant. We refer to a past definition of an operator by using a special temporal operator denoted *at* which extends the *pre* operator of `Lustre` [7] and *$* operator of `Signal` [9]. We write $at\,(-n\ \mathrm{s}\,)\ \omega$ to refer to the operator ω at the instant corresponding to n ticks of signal s before the current instant. In general, we can use several `at`-expressions to define the right instant in the past of an operator. For that we use *timed operator expressions* or *toe* for short, as defined below:

$$\phi \ ::= \ \omega \mid at\,(-n\ \mathrm{s}\,)\ \phi$$

where ω is an operator, n is a positive natural number, s is a signal, ϕ is a toe. The toes of the form $at\,(-n\ \mathrm{s}\,)\ \phi$ are called past toes. *Timed terms* extend classical ones by referring to past definitions of operators. They are defined as follows :

$$tt \ ::= \ x \mid \phi\ (\,u_1\,,\ldots,u_m\,)$$

where x is a variable, m is a natural number ($m \geq 0$), ϕ is a toe and $u_1\,,\ldots,u_m$ are well-sorted timed terms.

Formulas in R are timed conditional rewrite rules of the following form:

$$lhs \rightarrow trhs \Leftarrow B \ \textbf{when}\ C$$

where lhs is a first order term, $trhs$ is a timed term, B and C are possibly empty conjunctions of timed atoms. $lhs \rightarrow trhs$ is called *head*, B *body* and C *tail*. A timed atom is either an equation $u_1 == u_2$ where u_1 and u_2 are two timed terms, a formula $\Diamond|_1\ b_1\ ..\ b_2\ |_2\ B'$ which holds whenever the timed atom B' is true at least once during the interval $|_1\ b_1\ ..\ b_2\ |_2$, or a formula $\Box|_1\ b_1\ ..\ b_2\ |_2\ B'$ which holds whenever the timed atom B' is true at every instant of the interval $|_1\ b_1\ ..\ b_2\ |_2$. Atoms of the form $\Box|_1\ b_1\ ..\ b_2\ |_2\ B'$ (respectively, $\Diamond|_1\ b_1\ ..\ b_2\ |_2\ B'$) are called *past-box atoms* (respectively, *past-diamond atoms*). $|_1$ and $|_2$ stand either for the symbol [or for the symbol] indicating if the bounds are included or not in the interval, and b_1 and b_2 represent pairs noted $-n$ s where n is a non null natural number and s is a signal. We also use "now" as a particular form of b_2 to indicate the current instant. The role of tails is to filter the rewrite rules to be considered at each instant. A timed program $\mathcal{P}$ generates an infinite sequence of classical (atemporal) rewrite systems, $\mathcal{F}_t$, also called *stores* in the sequel, as depicted in Figure 1. Intuitively, at each instant t, $\mathcal{F}_t$ includes all the rewrite rules φ such that φ **when** C is in $\mathcal{P}$ and C holds at t. Since the tail is a conjunction of timed atoms, an empty (or missing) tail is always considered as true.

We introduce now some technical definitions and notations which we use in the sequel. We write $\mathcal{F} \vdash B$ to note that atom B holds in the classical rewrite

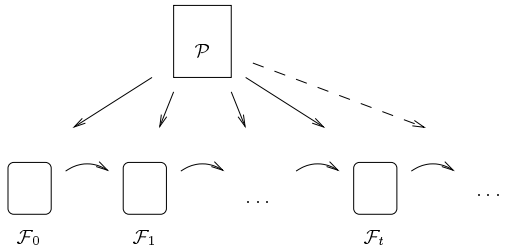


Fig. 1. Store Generation

system $\mathcal{F}$. We say that a timed rewrite rule ϕ *defines an operator* ω if ω is the outermost operator of the left-hand side of the head of ϕ. We note $\mathcal{D}ef_{\mathcal{P}}(\omega)$, the set of timed rewrite rules defining operator ω in the timed program $\mathcal{P}$. However, $\mathcal{D}ef_{\mathcal{P}}(\omega)$ is not self-contained in the sense that it may contain some operator ω' which is not defined in it. So, we introduce the notation $\mathcal{D}ef^c_{\mathcal{P}}(\omega)$ for the set of timed rewrite rules defining completely an operator ω in the timed program $\mathcal{P}$. For each operator ω' occurring in $\mathcal{D}ef^c_{\mathcal{P}}(\omega)$, $\mathcal{D}ef^c_{\mathcal{P}}(\omega)$ contains the timed rewrite rules of $\mathcal{P}$ defining ω'. We call *atemporal operator*, a timed operator ω of a timed program $\mathcal{P}$ such that $\mathcal{D}ef^c_{\mathcal{P}}(\omega)$ does not contain any reference to timed syntactic entities. Notice that a classical term rewrite systems consists only of atemporal operator definitions. We note $R\{\omega \longmapsto \omega'\}$ for the result of the replacement of a timed operator ω by a timed operator ω' in the set R of timed rewrite rules.

Below we give some toy examples in order to illustrate our framework.

Example 2.1 (Fibonacci)

In this example, we give two different definitions of Fibonacci function. This function is often used to show the abilities of synchronous languages. In our framework, we can define Fibonacci function either by following synchronous style (fib1) or a pure declarative style (fib2). However, fib1 will be a timed operator whose value will change during the execution and fib2 will be an atemporal operator.

1. fib1 $\rightarrow$ 0 **when** Ref == 0
2. fib1 $\rightarrow$ 1 **when** Ref == 1
3. fib1 $\rightarrow$ at (−1 Ref) fib1 + at (−2 Ref) fib1
   when Ref> 1 == true

1. fib2(0) $\rightarrow$ 0
2. fib2(1) $\rightarrow$ 1
3. fib2(x) $\rightarrow$ fib2(x-1) + fib2(x-2) $\Leftarrow x > 1 ==$ true

Example 2.2 (Steam boiler)

This example is inspired from [4]. The aim is to maintain the water level in a steam boiler between two limits low_level and high_level as far as possible. Without any intervention, this level is decreasing because of the outgoing of steam. The value of the nat-sorted incoming signal steam provides the current

measured steam, and the value of the nat-sorted incoming signal level provides the current measured level of water. If the level is out of the interval [low_level ; high_level] during more than 5 instants, an alarm must be triggered. The actual alarm is triggered whenever a new tick of boolean signal alarm occurs. The definition of alarm is:

alarm $\rightarrow$ true **when** $\Box$ [$-5\,\mathrm{REF}$ ..*now* [level $>$ high_level $==$ true
alarm $\rightarrow$ true **when** $\Box$ [$-5\,\mathrm{REF}$ ..*now* [level $<$ low_level $==$ true

Notice that the value of alarm is always true. This is not a problem since we are just interested in the ticks of the signal alarm, that is to say, the value of !alarm. This value equals true only if one of the two tails above holds.

The control of the water level is performed thanks to four pumps pump_1, pump_2, pump_3 and pump_4 of the same outflow P. pump_1, pump_2, pump_3 and pump_4 are atemporal operators of sort pump_sort, and P is a nat-sorted atemporal operator. These pumps feed the steam boiler with water in order to make up for the outgoing of water steam. They are controlled by a signal pump_order of profile pump_sort $\longrightarrow$ pump_order_sort. The data type pump_order_sort consists of two constructors, say on and off. When a tick of pump_order occurs, orders are sent to the pumps. pump_order (P) $==$ on means that pump p must be working and pump_order (P) $==$ off means that pump p must be idle. We suppose the orders are executed before the next instant without failure. In addition, work should be fairly distributed among pumps. Therefore, the value of pump_order depends on two parameters: the required number req_pump_nb of pumps to be "on" as well as the condition of fairness among the pumps. The last parameter is achieved by an ordering over the pumps given by a list (of pumps) ord_pump_list. The first elements of such a list represent the first pumps to be selected to perform the needed work at the next instant. ord_pump_list is partitioned into two lists working_pumps and resting_pumps such that working_pumps (resp. resting_pumps) is the list of the pumps which should be working (resp. idle) at the next instant. ord_pump_list is a signal of profile pump_sort list. working_pumps and resting_pumps are timed operators of profile pump_sort list. req_pump_nb is a timed operator of profile nat.

Whenever the list working_pumps is modified, we send the order on to each pump belonging to working_pumps and off to the others. So, the definition of pump_order is:

pump_order (p) $\rightarrow$ if is_in(p,working_pumps) then on else off
when working_pumps $\neq$ at $(-1\,\mathrm{REF})$ working_pumps $==$ true

The list of pumps working_pumps is the prefix of ord_pump_list such that the length of this prefix is the required number of pumps req_pump_nb. Conversely, the list of pumps resting_pumps is the remaining complementary list of pumps in ord_pump_list w.r.t. working_pumps. The definitions of working_pumps and

resting_pumps are (operators prefix and delete_prefix are straightforward):

working_pumps → prefix(req_pump_nb,ord_pump_list)
  when !ord_pump_list == true
resting_pumps → delete_prefix(req_pump_nb,ord_pump_list)
  when !ord_pump_list == true

On the other hand, the value of req_pump_nb depends on the measured quantity of steam (steam) as well as the measured water level (level). When level is greater than high_level, we need to lower the water level. So, the required number req_pump_nb of working pumps is the result of the division of steam over the outflow P. When level is lower than low_level, we need to rise the water level. So, the required number req_pump_nb of working pumps is the result of the division of steam by P, incremented by one. Thus, as the formulas calculating the number of required pumps are different according to the position of the level of water, we use a timed operator req_pump_fct of profile nat ⟶ nat which selects, at every instant, the right formula to be considered.

req_pump_nb → req_pump_fct (steam)

req_pump_fct (s) → s div P **when** level > high_level == true
req_pump_fct (s) → (s div P)+1 **when** level < low_level == true

Note that the following definition induces a way of calculating the required number of pumps different from the one of [4], since req_pump_nb is calculated to nearest penny at each instant even if level is between low_level and high_level.

The strategy we use to ensure a fair distribution of work among pumps consists in using pumps in turn for a fixed duration d where d is a natural number. The pumps which have worked more (resp. less) than d instants are said *strained* (resp. *fresh*). The list of strained, (resp. fresh) pumps is calculated by applying to working_pumps, the function strained (resp. fresh). These functions are timed operators of profile pump_sort list ⟶ pump_sort list.

Since the pumps react to orders immediately, the current state of a pump is given by the last order we sent. So, strained extracts from a pump list, the sublist of the pump p such that $\Box\,[\,-d\ \text{Ref}\ ..now\,[$ pump_order (p) == on.

strained ([]) → []
strained (p::l) → p::strained (l) ⇐
  $\Box\,[\,-d$ Ref $..now\,[$ pump_order (p) == on
strained (p::l) → strained (l) ⇐
  $\Diamond\,[\,-d$ Ref $..now\,[$ pump_order (p) == off

Likewise, fresh extracts from a pump list, the sublist of the pump p such

that $\Diamond\,[\,-d\,\text{Ref}\,..now\,[$ pump_order (p) == off:

fresh ([]) → []
fresh (p::l) → p::fresh (l) ⇐ $\Diamond\,[\,-d\,\text{Ref}\,..now\,[$ pump_order (p) == off
fresh (p::l) → fresh (l) ⇐ $\Box\,[\,-d\,\text{Ref}\,..now\,[$ pump_order (p) == on

Now we are ready to define ord_pump_list. It is built by appending (+) the following three lists:

- fresh (at $(-1\,\text{Ref})$ working_pumps) which stands for the still fresh working pumps,
- at $(-1\,\text{Ref})$ resting_pumps which stands for the current idle pumps,
- strained (at $(-1\,\text{Ref})$ working_pumps) which stands for the current strained pumps.

The updating of ord_pump_list happens when the number of pumps which we need to set on is modified (i.e., req_pump_nb $\neq$ at $(-1\,\text{Ref})$ req_pump_nb == true) or some working pumps are strained (i.e., strained (at $(-1\,\text{Ref})$ working_pumps) $\neq$ [] == true).

ord_pump_list → [pump_1 ; pump_2 ; pump_3 ; pump_4] **when** Ref == 0
ord_pump_list → fresh (at $(-1\,\text{Ref})$ working_pumps)
+ at $(-1\,\text{Ref})$ resting_pumps + strained (at $(-1\,\text{Ref})$ working_pumps)
when req_pump_nb $\neq$ at $(-1\,\text{Ref})$ req_pump_nb
or (strained (at $(-1\,\text{Ref})$ working_pumps)) $\neq$ [] == true

3 Labelled Transition System

We assume in the sequel that a timed rewrite system $\mathcal{P}$ is given with signature $< S\,,\,\mathcal{S}\,,\,\Omega >$. As depicted in Figure 1, a run of a program $\mathcal{P}$ generates a stream of stores, $\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_t$, *etc.* In this section, we define precisely how such stores are computed. A program $\mathcal{P}$ defines a labelled transition system. To each instant t, corresponds a state of such a transition system which contains, among other technical entities, a classical term rewrite system $\mathcal{F}_t$. Roughly speaking, the computation of $\mathcal{F}_t$ is done in four stages: $\mathcal{P}^\alpha$, $\mathcal{P}^\beta$, $\mathcal{P}^\gamma$ and $\mathcal{P}^\delta$. We explain these stages in the following.

3.1 Past toe and past atom replacement: $\mathcal{P}^\alpha$

$\mathcal{P}^\alpha$ is obtained from $\mathcal{P}$, at instant t, by eliminating toes and past atoms. This calculus is performed by a function $\langle . \rangle^t$ depending on time instant t and defined on the structure of the timed rewrite rules, so that, at a given instant t, $\mathcal{P}^\alpha = \langle \mathcal{P} \rangle^t$. Every toe occurring in $\mathcal{P}$ of the form $at\,(-n\,\text{s}\,)\,\omega$ is replaced by a new operator $\lfloor \omega \rceil^{t'}$ where t' denotes the number of the absolute instant, i.e. the value of Ref, at the instant corresponding to $at\,(-n\,\text{s}\,)$ w.r.t. to instant t.

t' is denoted by the expression $abs\_instant(\, n\,,\, \mathrm{s}\,,\, t\,)$ and satisfies the following property:

Property 1

- *At the instant t', a tick of* s *occurred.*
- *Exactly n ticks of* s *had happened from the instant t' until $t-1$.*

Thus, if t is the current instant, $abs\_instant(\, n\,,\, \mathrm{s}\,,\, t\,)$ is the instant which happened n ticks of s ago. The function $abs\_instant$ is partial. Indeed, $abs\_instant(\, n\,,\, \mathrm{s}\,,\, t\,)$ is not defined if less than n ticks of s had occurred from the first instant 0 until $t-1$. If a past toe $at\,(-n\,\mathrm{s}\,)\,\omega$ occurs in a timed rewrite rule and $abs\_instant(\, n\,,\, \mathrm{s}\,,\, t\,)$ is not defined, then $at\,(-n\,\mathrm{s}\,)\,\omega$ cannot be evaluated and the whole timed rewrite rule does not occur in $\mathcal{P}^{\alpha}$. The computation of $abs\_instant(\, n\,,\, \mathrm{s}\,,\, t\,)$ is performed by scanning the stores of the past from $t-1$ backwards until an instant t' verifying Property 1. $abs\_instant$ is formally defined by the following rules:

Definition 3.1 ($abs\_instant$)

(i) $abs\_instant(\, 1\,,\, \mathrm{s}\,,\, t\,) = t-1$
if $t>0$ and $\mathcal{F}_{t-1} \vdash !\mathrm{s} == \mathsf{true}$

(ii) $abs\_instant(\, n\,,\, \mathrm{s}\,,\, t\,) = abs\_instant(\, n-1\,,\, \mathrm{s}\,,\, t-1\,)$
if $abs\_instant(\, n-1\,,\, \mathrm{s}\,,\, t-1\,)$ is defined, $t>0$, $n>1$ and $\mathcal{F}_{t-1} \vdash !\mathrm{s} == \mathsf{true}$

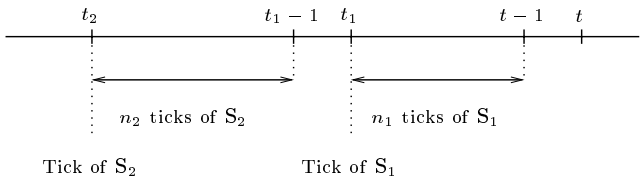
(iii) $abs\_instant(\, n\,,\, \mathrm{s}\,,\, t\,) = abs\_instant(\, n\,,\, \mathrm{s}\,,\, t-1\,)$
if $abs\_instant(\, n\,,\, \mathrm{s}\,,\, t-1\,)$ is defined, $t>0$, and $\mathcal{F}_{t-1} \vdash !\mathrm{s} == \mathsf{false}$

Rule (i) stops the scanning of the stores of the past, because $t-1$ verifies Property 1. Rule (ii) and Rule (iii) continue the scanning taking into account the possible occurrence of a tick of signal s at the previous instant. The proof that $abs\_instant(\, n\,,\, \mathrm{s}\,,\, t\,)$ verifies Property 1 is straightforward by using induction on the first argument of $abs\_instant$.

When a past toe is of the general form $at\,(-n_1\,\mathrm{s}_1\,)\,\ldots\,at\,(-n_m\,\mathrm{s}_m\,)\,\omega$, we proceed likewise by iterating recursively the function $abs\_instant$ to obtain the instant corresponding to $at\,(-n_1\,\mathrm{s}_1\,)\,\ldots\,at\,(-n_m\,\mathrm{s}_m\,)$.

Example 3.2 Consider Figure 2. To calculate t_2, the instant corresponding to $at\,(-n_1\,\mathrm{s}_1\,)\;at\,(-n_2\,\mathrm{s}_2\,)$ w.r.t. instant t, we need first to calculate instant t_1. t_1 corresponds to $at\,(-n_1\,\mathrm{s}_1\,)$ w.r.t instant t, i.e., $t_1 = abs\_instant\;(\, n_1\,,\, \mathrm{s}_1\,,\, t\,)$. Then, t_2 is obtained from t_1 as follows : $t_2 = abs\_instant\;(\, n_2\,,\, \mathrm{s}_2\,,\, t_1\,)$.

The elimination of a past atom, say B, of the form $\Diamond\,|_1 - n_1\;\mathrm{s}_1\;..-\,n_2\;\mathrm{s}_2|_2\;B'$ consists in replacing B by a disjunction $\bigvee_{t'\in|_1\,t_1\,;\,t_2\,|_2}\langle B'\rangle^{t'}$ where $t_1 = abs\_instant\;(\, n_1\,,\, \mathrm{s}_1\,,\, t\,)$ and $t_2 = abs\_instant\;(\, n_2\,,\, \mathrm{s}_2\,,\, t\,)$. If $abs\_instant$ is not defined on $(\, n_1\,,\, \mathrm{s}_1\,,\, t\,)$, then we consider $t_1 = 0$. In addition, if $abs\_instant$ is not defined on $(\, n_2\,,\, \mathrm{s}_2\,,\, t\,)$, then B is replaced by an atom noted FALSE which is always false. If the disjunction is empty then we consider B is replaced by FALSE



t_2 is the instant corresponding to $at\,(-n_1\ s_1)\ \ at\,(-n_2\ s_2)$ w.r.t. instant t.

Fig. 2. Absolute instant

too. The case of the past-box atoms, $\Box\ |_1\ -n_1\ s_1\ ..\ -n_2\ s_2\ |_2\ B'$, is similar to past-diamond atoms: disjunctions become conjunctions and the atom FALSE becomes an atom noted TRUE which is always true.

Finally, the formal definition of $\langle . \rangle^t$ is given below. Recall that $\mathcal{P}^\alpha = \langle \mathcal{P} \rangle^t$. By abuse of notation we note $\langle X \rangle^t$ the application of $\langle . \rangle^t$ on X where X could be either, a program, a timed rewrite rule, a conjunction of atoms, an equation, a term or a toe.

Definition 3.3 ($\langle . \rangle^t$)

Let $\mathcal{P}$ be a program and t an instant. $\mathcal{P}^\alpha = \langle \mathcal{P} \rangle^t$ is defined as follows.

-on sets of timed rewrite rules

$$\boxed{\langle \mathcal{P} \rangle^t = \{\ \langle \rho \rangle^t\ /\ \rho \in \mathcal{P}\ \}}$$

-on timed rewrite rules

$$\boxed{\begin{array}{l} \langle lhs \to trhs\ \Leftarrow\ B\ \textbf{when}\ C \rangle^t \\ \qquad = lhs \to \langle trhs \rangle^t\ \Leftarrow\ \langle B \rangle^t\ \textbf{when}\ \langle C \rangle^t \end{array}}$$

-on conjunctions of timed atoms

$$\boxed{\langle B \wedge B' \rangle^t = \langle B \rangle^t \wedge \langle B' \rangle^t}$$

- on equations

$$\boxed{\langle u_1 == u_2 \rangle^t = \langle u_1 \rangle^t == \langle u_2 \rangle^t}$$

- on past box atom of the form $B = \Box\ |_1 - n_1\ s_1\ ..\ - n_2\ s_2 |_2\ B'$

$\langle B \rangle^t = \bigwedge_{t' \in \|_1\ t_1\ ;\ t_2\ \|_2} \langle B' \rangle^{t'}$	if	$t_1 = abs\_instant(\ n_1\ ,\ s_1\ ,\ t\ )$ $t_2 = abs\_instant(\ n_2\ ,\ s_2\ ,\ t\ )$
$\langle B \rangle^t = \bigwedge_{t' \in [\ 0\ ;\ t_2\ \|_2} \langle B' \rangle^{t'}$	if	$abs\_instant$ is not defined on $(\ n_1\ ,\ s_1\ ,\ t\ )$ $t_2 = abs\_instant(\ n_2\ ,\ s_2\ ,\ t\ )$
$\langle B \rangle^t = \textsf{TRUE}$	if $abs\_instant$ is not defined on $(\ n_2\ ,\ s_2\ ,\ t\ )$	

- on past diamond atom of the form $B = \Diamond\ |_1 - n_1\ s_1\ ..\ - n_2\ s_2 |_2\ B'$

$\langle B\rangle^t = \bigvee_{t' \in \lfloor 1\ t_1\ ;\ t_2\ \rfloor_2} \langle B'\rangle^{t'}$	if $\begin{cases} t_1 = abs\_instant(\ n_1\ ,\ \mathrm{s}_1\ ,\ t\ ) \\ t_2 = abs\_instant(\ n_2\ ,\ \mathrm{s}_2\ ,\ t\ ) \end{cases}$
$\langle B\rangle^t = \bigvee_{t' \in \lfloor 0\ ;\ t_2\ \rfloor_2} \langle B'\rangle^{t'}$	if $\begin{cases} abs\_instant \text{ is not defined on } (\ n_1\ ,\ \mathrm{s}_1\ ,\ t\ ) \\ t_2 = abs\_instant(\ n_2\ ,\ \mathrm{s}_2\ ,\ t\ ) \end{cases}$
$\langle B\rangle^t = \mathsf{FALSE}$	if $abs\_instant$ is not defined on $(\ n_2\ ,\ \mathrm{s}_2\ ,\ t\ )$

- on timed terms

$\langle x\rangle^t = x$
$\langle \phi(\ u_1\ ,\ldots,\ u_m\ )\rangle^t = \langle\phi\rangle^t(\ \langle u_1\rangle^t\ ,\ \ldots\ ,\ \langle u_m\rangle^t\ )$

- on toes

$\langle\omega\rangle^t = \lfloor\omega\rceil^t$	if ω is not atemporal
$\langle\omega\rangle^t = \omega$	if ω is atemporal
$\langle at\,(-n\ \mathrm{s}\ )\,\phi\rangle^t = \langle\phi\rangle^{abs\_instant(\ n\ ,\ \mathrm{s}\ ,\ t\ )}$	if $abs\_instant$ is defined on $(\ n\ ,\ \mathrm{s}\ ,\ t\ )$

Where ω is a timed operator, ϕ a toe, s a signal, x a variable, lhs a first-order term, $trhs$, $u_1\ ,\ldots,\ u_{m-1}$ and u_m timed terms, B, B' and C timed atoms and $m \geq 0$, $n > 0$.

Example 3.4 Consider the following toy example:

1. f(x) $\rightarrow$ x+1 **when** s $>$ 15 == true
2. f(x) $\rightarrow$ x-1 **when** s $<$ 5 == true
3. m $\rightarrow$ 10 **when** Ref == 0
4. m $\rightarrow$ f(at (−1 s) m) **when** $\Box$ [-3 Ref .. -1 Ref] s $\leq$ 10 == true

where s : nat is an incoming signal, f : nat $\longrightarrow$ nat is a timed operator, m : nat is a signal.

Assume that a tick of s occurs at instant 0 and does not occur at instant 1. Thus, at instant 2, $\mathcal{P}^\alpha$ for this specification is:

$\langle 1\rangle^2$. $\lfloor$f$\rceil^2$ (x) $\rightarrow$ x+1 **when** $\lfloor$s$\rceil^2 > 15$ == true
$\langle 2\rangle^2$. $\lfloor$f$\rceil^2$ (x) $\rightarrow$ x-1 **when** $\lfloor$s$\rceil^2 < 5$ == true
$\langle 3\rangle^2$. $\lfloor$m$\rceil^2 \rightarrow$ 10 **when** $\lfloor$Ref$\rceil^2$ == 0
$\langle 4\rangle^2$. $\lfloor$m$\rceil^2 \rightarrow \lfloor$f$\rceil^2$ ($\lfloor$m$\rceil^0$) **when** $\lfloor$s$\rceil^0 \leq 10$ == true $\wedge\ \lfloor$s$\rceil^1 \leq 10$ == true

At instant 0, $\mathcal{P}^\alpha$ is:

$\langle 1\rangle^0$. $\lfloor$f$\rceil^0$ (x) $\rightarrow$ x+1 **when** $\lfloor$s$\rceil^0 > 15$ == true
$\langle 2\rangle^0$. $\lfloor$f$\rceil^0$ (x) $\rightarrow$ x-1 **when** $\lfloor$s$\rceil^0 < 5$ == true
$\langle 3\rangle^0$. $\lfloor$m$\rceil^0 \rightarrow$ 10 **when** $\lfloor$Ref$\rceil^0$ == 0

Note that $\langle 4\rangle^0$ does not hold in $\mathcal{P}^\alpha$ because $\langle$at (−1 s) m$\rangle^0$ is not defined.

3.2 Adding needed definitions: $\mathcal{P}^\beta$

In the second stage, we compute a new program $\mathcal{P}^\beta$. $\mathcal{P}^\beta = \mathcal{P}^\alpha \cup In \cup D$ where In is the set of rules defining the incoming signals and D consists of rewrite rules defining the new operators introduced in $\mathcal{P}^\alpha$, of the form $\lfloor\omega\rceil^{t'}$, where $0 \leq t' < t$. The axioms in D are copied from the stores previously computed. At the end of this stage, $\mathcal{P}^\beta$ is a timed program without any reference to toes nor past atoms.

Example 3.5 Consider the timed program given in Example 3.4. Assume that a tick of s occurs at instant 0 and at instant 2, but does not occurs at instant 1. Suppose also that at instant 0, the given value of signal s is 2 and at instant 2, the given value of s is 8. So, at instant 2, the rewrite rules defining the incoming signals (In) are:

$$5.\ \lfloor \text{Ref} \rceil^2 \to 2 \qquad 6.\ \lfloor \mathsf{s} \rceil^2 \to 8$$

On the other hand, we need the definitions of operators $\lfloor \mathsf{m} \rceil^0$, $\lfloor \mathsf{s} \rceil^0$, $\lfloor \mathsf{s} \rceil^1$ to complete $\mathcal{P}^\alpha$. These definitions (D), given below, are available in the stores $\mathcal{F}_0$ and $\mathcal{F}_1$.

$$7.\ \lfloor \mathsf{m} \rceil^0 \to 2 \qquad 8.\ \lfloor \mathsf{s} \rceil^0 \to 2 \qquad 9.\ \lfloor \mathsf{s} \rceil^1 \to 2$$

So, at instant 2, $\mathcal{P}^\beta$ consists of all the timed rewrite rules of $\mathcal{P}^\alpha$ and the rewrite rules 5 to 9. Note that, at instant 0, the set of definitions D of the new operators introduced in $\mathcal{P}^\alpha$ is empty because no instant happened before instant 0.

3.3 Tail removal: $\mathcal{P}^\gamma$

At a third stage, we compute a new program $\mathcal{P}^\gamma$ from $\mathcal{P}^\beta$. Roughly speaking, $\mathcal{P}^\gamma$ contains the rewrite rules φ such that there exists a timed rewrite rule "φ **when** C" in $\mathcal{P}^\beta$ where C holds in $\mathcal{P}^\beta$ (cf. Definition 3.7 for a precise definition). Whenever all the tails of rules defining an operator $\lfloor\omega\rceil^t$ in $\mathcal{P}^\beta$ are false, we define $\lfloor\omega\rceil^t$ in $\mathcal{P}^\gamma$ as $\lfloor\omega\rceil^{t-1}$. This last operation enables the timed operators to have a value even if they are not redefined at the current instant (remanence property). At instant t, if $\lfloor\omega\rceil^t$ represents the current value of some signal s and if $\lfloor\omega\rceil^t$ has been redefined, then a tick of s occurs and the rule !s $\to$ *true* is added to $\mathcal{P}^\gamma$. Otherwise, no tick of s occurs and the rule !s $\to$ *false* is added to $\mathcal{P}^\gamma$.

Example 3.6 Consider the timed program given in Example 3.4 with the assumptions of Example 3.5.

At instant 2, since the value of s is 8, neither the tail of $\langle 1 \rangle^2$, nor the tail of $\langle 2 \rangle^2$ holds in $\mathcal{P}^\beta$. Thus, the definition of $\lfloor \mathsf{f} \rceil^2$ is the one of $\mathcal{F}_1$ up to renaming of the operators. Assume the timed rewrite rule defining $\lfloor \mathsf{f} \rceil^1$ in $\mathcal{F}_1$ is $\lfloor \mathsf{f} \rceil^1$ (x) $\to$ x-1. Hence, the definition of $\lfloor \mathsf{f} \rceil^2$ occurring in $\mathcal{P}^\gamma$ is:

$1'.\ \lfloor \mathsf{f} \rceil^2$ (x) $\to$ x-1

Likewise, the tail $\lfloor \text{Ref} \rceil^2 == 0$ of $\langle 3 \rangle^2$ does not hold in $\mathcal{P}^\beta$. However, the tail $\lfloor \text{s} \rceil^0 \leq 10 == \text{true} \wedge \lfloor \text{s} \rceil^1 \leq 10 == \text{true}$ of $\langle 4 \rangle^2$ is true in $\mathcal{P}^\beta$. Thus, the definition of $\lfloor \text{m} \rceil^2$ occurring in $\mathcal{P}^\gamma$ is:

$2'$. $\lfloor \text{m} \rceil^2 \rightarrow \lfloor \text{f} \rceil^2 \ (\lfloor \text{m} \rceil^0)$

Since m is a signal, $\mathcal{P}^\gamma$ also contains:

$3'$. $\lfloor !\text{m} \rceil^2 \rightarrow \text{true}$

The timed rewrite rules 5 to 9 are classical rewrite rules. Hence, they belong to $\mathcal{P}^\gamma$, straightforward. Since Ref and s are signals, the following timed rewrite rules defines their tick:

10. $\lfloor !\text{Ref} \rceil^2 \rightarrow \text{true}$
11. $\lfloor !\text{s} \rceil^2 \rightarrow \text{true}$

So, $\mathcal{P}^\gamma$ consists of the rewrite rules $1'$, $2'$, $3'$ and the rewrite rules 5 to 11.

Since all the timed rewrite rules are considered at the same instant, the deletion of their tails may be not computable due to causality loops. The following example illustrates the problem:

$A \rightarrow \text{true}$ **when** $B == \text{true}$

$B \rightarrow \text{true}$ **when** $A == \text{true}$

The computation of the definition of $\lfloor A \rceil^t$ requires the computation of the definition of $\lfloor B \rceil^t$ and vice versa. Hence, the elimination of tails is not possible in this case. The problem of causality loop exists in most of synchronous programming languages [10]. A solution consists in stratifying the considered program, i.e., finding an ordered partition of the set of the timed operators into subsets, say *strata*, such that:

Requirement 1 *The tails of each timed rewrite rule of a given stratum are closed and decidable formulas expressed through the operators of the lower strata.*

The order relation over the partition is partial. To make easier the reading, we linearize it so to be able to number the strata. All the strata are noted $stratum_i$ where i is a natural number between 1 and the cardinal n_{strata} of the partition. They are indexed according to the order on strata from the lowest to the highest. Notice that such stratification of programs has also been considered in logic programming languages (e.g. [13]). In addition, for real-time applications, tails should be computable in a sensible amount of time.

The stratification induces a data-flow calculus for the construction of $\mathcal{P}^\gamma$. The formal definition of $\mathcal{P}^\gamma$ is given in Definition 3.7. However, we need to introduce first a technical definition which represents a rough tail deletion.

$$\mathcal{V}alid\_def_{\mathcal{P},\mathcal{F}}(\omega) = \{ \ \varphi \ / \ \text{“}\varphi \ \textbf{when} \ C\text{”} \in \mathcal{D}ef_{\mathcal{P}}(\omega) \text{ and } \mathcal{F} \vdash C \ \}$$

where $\mathcal{P}$ is a timed program and $\mathcal{F}$ is a classical first-order program.
$\mathcal{V}alid\_def_{\mathcal{P},\mathcal{F}}(\omega)$ consists of the rewrite rules φ such that a timed rewrite rule of the form φ **when** C belongs to $\mathcal{P}$ and C is true in the first-order program $\mathcal{F}$.

Definition 3.7 ($\mathcal{P}^\gamma$)
Let $(\mathcal{P}^\gamma_0, \ldots, \mathcal{P}^\gamma_{n_{strata}})$ be a finite stream of classical first order program defined as follows:

- $\mathcal{P}^\gamma_0 = \emptyset$,
- $\mathcal{P}^\gamma_i = \mathcal{P}^\gamma_{i-1} \cup A_i \cup B_i \cup C_i \cup D_i$ for all $i \in [1, n_{strata}]$

where:

$$A_i = \bigcup_{\substack{\omega \in stratum_i \\ \forall \mathrm{S} \in \mathcal{S}, \omega \neq !\mathrm{S}}} \mathcal{V}alid\_def_{\mathcal{P}^\beta, \mathcal{P}^\gamma_{i-1}}(\lfloor \omega \rceil^t)$$

$$B_i = \bigcup_{\substack{\omega \in stratum_i \\ \forall \mathrm{S} \in \mathcal{S}, \omega \neq !\mathrm{S}}} \{\ \mathcal{D}ef_{\mathcal{F}_{t-1}}(\lfloor \omega \rceil^{t-1})\ \{\lfloor \omega \rceil^{t-1} \longmapsto \lfloor \omega \rceil^t\}\ /\ \mathcal{V}alid\_def_{\mathcal{P}^\beta, \mathcal{P}^\gamma_{i-1}}(\lfloor \omega \rceil^t) = \emptyset\ \}$$

$$C_i = \bigcup_{\mathrm{S} \in stratum_i \cap \mathcal{S}} \{\ !\mathrm{s} \rightarrow true\ /\ \mathcal{V}alid\_def_{\mathcal{P}^\beta, \mathcal{P}^\gamma_{i-1}}(\lfloor \mathrm{s} \rceil^t) \neq \emptyset\ \}$$

$$D_i = \bigcup_{\mathrm{S} \in stratum_i \cap \mathcal{S}} \{\ !\mathrm{s} \rightarrow false\ /\ \mathcal{V}alid\_def_{\mathcal{P}^\beta, \mathcal{P}^\gamma_{i-1}}(\lfloor \mathrm{s} \rceil^t) = \emptyset\ \}$$

Then, the result $\mathcal{P}^\gamma$ of the third stage is $\mathcal{P}^\gamma_{n_{strata}}$.

Since $\mathcal{P}^\gamma_0 \subseteq \mathcal{P}^\gamma_1 \subseteq \cdots \subseteq \mathcal{P}^\gamma_{n_{strata}}$, $\mathcal{P}^\gamma$ is constructed stepwise on the strata. At each stratum i, we add the definitions of timed operators contained in $stratum_i$. A_i (resp. B_i) represents the set of definitions of the timed operators (different from signal ticks) which have been re-defined (resp. have not been re-defined) at the current instant. C_i (resp. D_i) represents the definition of the signal ticks which have occurred (resp. have not occurred) at the current instant.

3.4 Rewrite rule simplification: $\mathcal{P}^\delta$

$\mathcal{P}^\gamma$ is a set of classical rewrite rules. However, we give the following example to motivate an ultimate stage of simplification. Suppose that, at each instant t, $\mathcal{F}_t$ is calculated from the first three stages $\mathcal{P}^\alpha$, $\mathcal{P}^\beta$ and $\mathcal{P}^\gamma$. Consider the timed program given in Example 3.4 with the assumptions of Example 3.5. $\mathcal{F}_3$ contains the rewrite rule$\lfloor \mathsf{m} \rceil^3 \rightarrow \lfloor \mathsf{f} \rceil^3\ (\lfloor \mathsf{m} \rceil^2)$ and the complete definition of $\lfloor \mathsf{m} \rceil^2$ taken from $\mathcal{F}_2$ (i.e. $\mathcal{P}^\gamma$ in Example 3.6). This complete definition of $\lfloor \mathsf{m} \rceil^2$ is:

2'. $\lfloor \mathsf{m} \rceil^2 \rightarrow \lfloor \mathsf{f} \rceil^2\ (\lfloor \mathsf{m} \rceil^0)$
1'. $\lfloor \mathsf{f} \rceil^2\ (\mathsf{x}) \rightarrow \mathsf{x\text{-}1}$
7. $\lfloor \mathsf{m} \rceil^0 \rightarrow 2$

So, one may guess that, the number of rewrite rules needed to define $\lfloor m \rceil^t$ increases at each instant t. This is not tractable. So, we add a fourth stage whose aim is to prevent from unbound increase of the number of timed rewrite rules in the stores during the execution. It consists in computing for each timed constant [1] ω, all its values v and replacing the definition of ω by atemporal

[1] i.e. timed operator without arguments

rules $\omega \to v$. The new program is noted $\mathcal{P}^\delta$, and at the instant t, $\mathcal{F}_t$ is $\mathcal{P}^\delta$.

Example 3.8 Consider $\mathcal{P}^\gamma$ in Example 3.6. The rewrite rule $1'$ defines a function. Thus, it remains unchanged. The rewrite rule $3'$ and the rewrite rules 5 to 11 define constants. However, since their right-hand side is in normal form, they cannot be simpler. On the other hand, the remaining rewrite rule $2'$ defines a constant and can be simplified. Since the normalization of $\lfloor \mathsf{m} \rceil^2$ leads to 1, the new rewrite rule replacing $2'$ is $\lfloor \mathsf{m} \rceil^2 \to 1$.

Unfortunately, the stage of simplification cannot be applied to timed functions[2] in the general. For example, an operator, f, defined by $\mathsf{f(x)} \to \mathsf{at}\,(-1\,\textsc{Ref})\ \mathsf{f(x)} + \mathsf{g(x)}$ where g is an input signal, needs at every instant, t, all the rewrite rules which have been used to define $\lfloor \mathsf{f} \rceil^{t'}$ since the first instant, i.e., for $0 \leq t' < t$. The simplification would consist in searching all the values of ω for each possible values of the arguments, which is not tractable. So, we assume in the sequel, the following requirement:

Requirement 2

(i) *A timed constant ω must be recursive*[3]*, in $\mathcal{P}^\gamma$, at each instant.*

(ii) *A timed function ω cannot be defined recursively from the past, in $\mathcal{P}^\gamma$, at each instant.*

For instance, the operator, f, of Example 3.4 satisfies Requirement 2 because it does not depend on any timed operator whose definitions change over time. Likewise, m is recursive, and thus it satisfies Requirement 2. The last item (ii) may be weakened because a timed function which is defined over a finite domain can be considered as a finite array of timed constants. Hence, the scope of item (ii) can be reduced to set of functions with an infinite domain.

3.5 Labelled transition system

$\mathcal{P}^\alpha$, $\mathcal{P}^\beta$, $\mathcal{P}^\gamma$ and $\mathcal{P}^\delta$ are the four stages leading to the construction of the store $\mathcal{F}_t$ at a given instant t. The following labelled transition system indicates from a more global point of view how the stores will be generated over time during the execution of the timed program.

Definition 3.9 (labelled transition system)
Let $\mathcal{P}$ be a timed program. The associated *labelled transition system* is a tuple $< Q\ ,\ L\ ,\ \longrightarrow\ ,\ Init >$ such that :

- Q is the set of *states*. A state is a pair $< \mathcal{H}\ ,\ \mathcal{F} >$ where $\mathcal{H}$, called *history*, is a list of stores and $\mathcal{F}$ is a store. The stores define at different instants operators occurring in $\mathcal{P}$.
- L is the set of *labels* where a label is a set of rules defining the incoming signals.

[2] i.e. timed operator with arguments
[3] i.e. computable and terminating

- $\longrightarrow \subseteq Q \times L \times Q$ is the transition relation defined as follows. We write $\mathcal{C}_1 \xrightarrow{In} \mathcal{C}_2$ for a triple $<\mathcal{C}_1\,,\,In\,,\,\mathcal{C}_2> \in \longrightarrow$. A transition $<\mathcal{H}\,,\,\mathcal{F}_t> \xrightarrow{In} <[\mathcal{F}_t \mid \mathcal{H}]\,,\,\mathcal{F}_{t+1}>$ occurs iff $\mathcal{F}_{t+1}$ is the store computed at the instant $t+1$ starting from the program $\mathcal{P}$, the history $[\mathcal{F}_t \mid \mathcal{H}]$ and the input signals given by In.
- $Init$ is a function $L \longrightarrow Q$ of *initialization* such that if In_0 represents the definitions of the incoming signals at instant 0, then $Init(In_0) = <[]\,,\,\mathcal{F}_0>$.

A state, $<\mathcal{H}\,,\,\mathcal{F}>$, is composed of the current store, $\mathcal{F}$, and the history, $\mathcal{H}$, a list of the previous stores. Notice that the computation of $\mathcal{F}$ may need some definitions from old stores we can find in $\mathcal{H}$.

4 Related work

Synchronous languages are specially designed for reactive programming [10]. The most representative synchronous languages are `Lustre` [7], `Signal` [9] and `Esterel` [2]. All of them are based on the notion of signal and compile programs into finite state automata. However, these languages do not handle new abstract data types or functions as signals. So, needed_pump_function cannot be defined as directly as in Example 2.2. Moreover, they do not provide powerful temporal operators such as $\Box$ and $\Diamond$ over intervals of time. The closest languages to our formalism are the declarative ones, namely `Lustre` and `Signal`, the primitives of which may be divided into three categories:

- *The classical arithmetic or logical operators* are static operators extended to streams. We can define them as atemporal operators.
- *The delay operators* (pre in `Lustre` and $ in `Signal`) allow to access during the execution to past values of signals. The operator `at` subsumes these operators.
- *The polychronous operators* allow one to express new signals from others having different clocks. This can be easily expressed in our formalism by means of tail formulas.

As for `Esterel`, it rather offers an imperative style of programming and, as in our semantics, signals are remanent unlike `Lustre` and `Signal`. Note that the compilation into a finite automaton is possible in our case too whenever the considered applications do not need more expressive machines.

The absence of time notion in concurrent constraint programming `ccp` [18] has led to the development of a new synchronous paradigm `tcc` [16] based on `ccp` and preserving its good properties. We share with `tcc` the principle of a very expressive evolving store. However, the way of expressing the store changes is completely different. In `tcc`, the stores are generated from processes and the process algebra is orthogonal to the formalism expressing formulas of the stores. Moreover, these formulas are atemporal and not remanent. Thus, the data transmission from an instant to another is performed explicitly by

the combinator next of the process algebra. Timed term rewrite systems are rather data-oriented, which implies that the mechanisms of data transmission are hidden from the user. A problem of tcc inherited from ccp is the negative information detection. At each instant, tcc may only detect an absence of information at the end of the store computation, and thus the reaction is put back one instant later. A solution was presented in [17]. The new paradigm is called Timed Default cc and enables to detect an absence of information and to react instantaneously. In compensation, the programs need to be verified statically to avoid causality loops, whereas it was not necessary for tcc programs ("paradox-free" property of [16]). Our store is a well-defined stratified rewrite system which may be adapted to a large scope of constraints. A boolean function may stand for a predicate solving the NOT problem in the logic programming paradigm and thus the negative information problem of ccp. On the other hand, ntcc [15] extends tcc by adding non-determinism. This paradigm has, among others, two operators noted ! and $\star$. !P means "P will be always true in the current store and in the future ones", $\star$P means "P will be true in the current store or in some future one". From a logic point of view, ! and $\star$ are the respective duals in the future of the operators $\Box$ and $\Diamond$. From a procedural point of view, ! and $\star$ can be considered as behavior generators, whereas $\Box$ and $\Diamond$ are behavior testers. ! and $\star$ can be applied just to a specific interval of time in future. However, this interval is expressed using the base time unit: no other granularity is provided.

Temporal logic programming languages [14] are also related to our formalism. The meaning of "temporal" for languages like Templog [1] and Temporal Prolog of Gabbay [8] is not the same as the one in the synchronous world. The time is indeed represented in a constraint system and does not constrain temporally the execution of the program. So, the causality of the condition in formulas is not preserved in these extensions of the logic programming, whereas it is in our paradigm. On the other hand, these languages have the temporal operators $\Box$ (always) and $\Diamond$ (eventually) without any reference to time intervals. Chronolog is a data-flow temporal logic language [19] designed to perform efficient computations. However, this language only manages one clock. Since the need for a multi-granular time is very important in temporal programming, Chronolog(MC) [12] has been developed. Nevertheless, this language is not synchronous.

5 Conclusion

We have introduced timed term rewrite systems as a conservative extension of classical term rewrite systems which allow one to write declarative programs where time is involved. We have also defined precisely the behavior of such rewrite systems by means of labelled transition systems. Such transition systems are not meant to define actual operational semantics. We quote here two main reasons for that:

- The history within the states of a transition system increases over time. Thus the implementation of the history is prohibited.
- The size of the rewrite rules in the different stores may increase over time too. Indeed, let init be a signal such that its tick occurs only at instant 0. If we consider now the past atom $B = \Box\,[\,-1\,\mathsf{init}\,..\,-1\,\textsc{Ref}\,]\,B'$ then $\langle B\rangle^3 = \langle B'\rangle^0 \wedge \langle B'\rangle^1 \wedge \langle B'\rangle^2$. Thus, the length of the conjunction will increase by one at each instant. Hence, the length of some timed rewrite rule containing this conjunction will grow at the same rate.

The reader interested in an efficient operational semantics may consult [3]. The proposed new semantics allows one to tackle real-world applications. The main change consists in replacing the history by a finite set of bounded memories.

References

[1] M. Abadi and Z. Manna. Temporal logic programming. *J. of Symbolic Computation*, 8(3):277–295, September 1989.

[2] G. Berry and G. Gonthier. The Esterel programming language: Design, semantics and implementation. *Science of Computer Programming*, 19(2), 1992.

[3] J. Blanc and R. Echahed. On the operational semantics of timed rewrite systems. In *9th Int. Symp. on Temporal Representation and Reasoning (Time'02)*, Manchester, UK, July 2002.

[4] T. Cattel and G. Duval. The steam boiler problem in lustre. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, LNCS 1165, 1996.

[5] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, chapter 6, pages 243 – 320. Elsevier, Amsterdam, 1990.

[6] N. Dershowitz and D. A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning, Volume I*, chapter 9, pages 535 – 610. Elsevier, Amsterdam, 2001.

[7] C. Dumas-Canovas and P. Caspi. A PVS proof obligation generator for Lustre programs. In *7th Int. Conf. on Logic for Programming and Automated Reasoning*, volume 1955 of *LNAI*, 2000.

[8] D. Gabbay. *Modal and Temporal Logic Programming*, chapter 6, pages 197–237. Academic Press, 1987.

[9] P. L. Guernic, T. Gautier, M. L. Borgne, and C. L. Maire. Programming real time applications with signal. *Proceedings of the IEEE*, 79(9), 1991.

[10] N. Halbwachs. Synchronous programming of reactive systems. In *Computer Aided Verification (CAV'98)*, LNAI 1427, pages 1–16, 1998.

[11] J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 1 – 112. Oxford University Press, 1992.

[12] C. Liu and M. A. Orgun. Dealing with multiple granularity of time in temporal logic programming. *J. of Symbolic Computation*, 22(5 and 6):699–720, 1996.

[13] J. W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation Artificial Intelligence series. Springer-Verlag, 1987.

[14] M. A. Orgun and W. Ma. An overview of temporal and modal logic programming. In D. M. Gabbay and H. J. Ohlbach, editors, *First Int. Conf. on Temporal Logic*, LNAI 827, pages 445–479, July 1994.

[15] C. Palamidessi and F. Valencia. A temporal concurrent constraint programming calculus. Report RS-01-20, BRICS, University of Aarhus, June 2001.

[16] V. Saraswat, R. Jagadeesan, and V. Gupta. *Constraint Programming*, volume 131 of *the NATO Advanced Science Institute Series, Series F: Computer and System Sciences*, chapter Programming in Timed Concurrent Constraint Languages. Springer Verlag, 1994.

[17] V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5 – 6):475 – 520, Nov – Dec 1996.

[18] V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proceedings of the* 17^{th} *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL'90)*, pages 232 – 245, San Francisco, Jan. 1990. extended abstract.

[19] K. Zhang and M. A. Orgun. Parallel execution of temporal logic programs using dataflow computation. In *Int. Conf. on Computing and Information*, pages 812–830, 1994.